AMENDMENTS TO THE SPECIFICATION:

Please amend the heading at page 1, line 4:

~~Chapter I: Introduction~~BACKGROUND

1. Technical Field

Please amend the heading at page 1, line 8:

~~Prior Art~~2. Related Art

Please amend the heading at page 1, line 26:

~~Invention~~BRIEF SUMMARY

Please add the following at page 1, line 27:

BRIEF DESCRIPTION OF THE DRAWINGS

Please amend the heading at page 2, line 17:

~~Chapter II: Infrastructure~~DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS

Please amend the paragraph at page 8, beginning at line 31:

Those skilled in the art will be aware that in human-human conversations the prosody of an utterance -- as well as grammatical patterns -- plays an important role in the timing of turn-taking between actors in a conversation. It is anticipated that speech recognition algorithms in the future will also routinely mimic this ability. For example the extremely rapid detection of the end of an STD code observed in human-human number transfers facilitated by grammatical knowledge of STD patterns, and also detected from the prosodic pattern of the chunking. "Is the speaker done yet? Faster and more accurate end-of-utterance detection using prosody" Ferrer, ~~Shirbirg~~Shirberg, Stolcke -- ISCA Workshop on Prosody and Speech Recognition describes one method of doing this. Addition of this feature to the invention will increase its power to emulate the chunked transfer strategies seen in human-human conversation.

Please amend the paragraph at page 9, beginning at line 8 as follows:

Spoken input during the echoing of a block is ignored. This is because it has been observed that givers tend to defer to the follower when overlap occurs and ignoring attempts at interruption helps to enforce the intended chunked echo protocol,--reducing the risk of confusing the giver. Some of the embodiments of the invention described facilitate chunked confirmation. This involves speaking a number back to a giver by splitting it into chunks. After each chunk is spoken, there is a pause for spoken input between the chunks. Readback is continued if no response is received (i.e. silence). This pattern could be viewed as reading out a single number sequence containing internal phrasal boundaries. If it is viewed in this way then interruption should be viewed as permitted in the regions around the phrasal boundaries[[.]] (i.e. in the phrasal pause between blocks). Slight overlap of the interruption and the number output could be permitted at the boundaries.

Please amend the heading at page 9, line 19:

~~Chapter HH:~~Dialogue, first version

Please amend the paragraph at page 11, beginning at line 17:

Block boundaries are stored as an array of **L** elements indexed from zero (i.e. $\mathbf{B}_0$, $\mathbf{B}_1$ . . . $\mathbf{B}_{L-1}$), where L is an arbitrary limit greater than the number of blocks which will be exchanged in a dialogue (e.g. 20 for telephone numbers, as blocks can be as small as one digit and there could be additional correction digits). The value of each entry in the array records the index in the telno buffer where a block has started. In the example, the region marked "confirmed" will have previously been output as a `current_block`, which started at telno index 0. Therefore block boundary zero points to location zero (i.e. $\mathbf{B}_0=\mathbf{0}$. The **current_block** shown in the figure starts at index 5, so $\mathbf{B}_1$=5. This is the last block boundary as it represents the start point of the **current_block** at this point in time. In the figures, the block boundaries will be shown as arrows pointing to the boundary to the left of the telno entry they are indicating.

Please amend the paragraph at page 13, beginning at line 1:

(a) Unclear digits. In case of an input_block entry that is unclear (e.g. "3 ? 4" with the middle digit being difficult to hear), the input_block is not added to telno, and a "sorry?" prompt is played ~~(215)~~(212) to prompt for a repetition of this block. (This mimics the use of "sorry" found in the operator dialogues.)

Please amend the paragraph at page 14, beginning at line 11:

Too many digits given. This may indicate that one of the blocks of digits (given under condition (f) above) was intended to replace, rather than follow, the digits previously recognised and echoed. To cope with this, a finalRepair algorithm is applied ~~224~~244 to the telno buffer. This final repair algorithm is described in detail below.

Please amend the paragraph at page 17, beginning at line 9:

It is possible to enforce a "chunked" dialogue style, with a low value for T.sub.STD (e.g. 0.4s) when an STD code is expected, or an "unchunked" style, with the T.sub.STD set to default T.sub.D or longer.

Please amend the heading at page 18, line 1:

~~Chapter IV:~~Dialogue, second version

Please amend the paragraph at page 18, beginning at line 6:

The dialogue with this third strategy follows that described with reference to FIG. 2 except when an input_block (given without an initial "no") contains more than five digits. When this happens, instead of the output being simply an echo of the whole current_block, a chunked confirmation sub-dialogue is entered, which is as shown in Figure 4. This sub-dialogue replaces the simple "play <current_block>" (210, 218, ~~238~~228) occurring in the basic dialogue and appearing in paths (c), (e) and (f) of FIG. 2. During the chunked confirmation sub-dialogue, successive chunks, typically containing three or four digits each, are echoed back to the giver; there are pauses between chunks, in which the giver can respond by confirming, contradicting, correcting or continuing the digit sequence just echoed. The first chunk in the sequence is preceded by "that's" if this is the first echo of a digit block in the current dialogue, or the first echo following a re-prompt for the whole code and number. The sub-dialogue ends with the output of the last chunk of the current_block, at which point the main dialogue resumes and possible inputs from the giver are dealt with as in Figure ~~II~~2

Please amend the paragraph at page 22, beginning at line 1:

The process of Figure 4 starts at Step 700. At Step 700, the values of $\mathbf{f_0}$ and $\mathbf{f_r}$ define the current_block in Figure ~~1~~2 and this current_block contains the last input which in the previous design would have been played out in full to the giver.

Please amend the paragraph at page 22, beginning at line 24:

At Step 703, *chunk* is played. If at Step 704, the remainder is null (i.e. $\mathbf{f_r=fg}$), control reverts (705) to Figure ~~1~~2 Recall at this point that current_block is synonymous with chunk because they both depend on $\mathbf{f_0}$ and $\mathbf{f_r}$. Thus control will continue in Figure ~~1~~2 now treating only the last **chunk** that was played as the current block.

Please amend the paragraph at page 22, beginning at line 28:

Otherwise, if remainder is not null, the input buffer is read at Step 706. Various types of non-digit input at 707 to 709, and 713 are dealt with much as in Figure ~~1~~2.

Please amend the paragraph at page 23, beginning at line 5:

Digits input at 710 or 711 are dealt with firstly (as in Figure ~~1~~2) by local repair if any internal correction at 714. It then moves on to the alignInput function of Step 715.

Please amend the paragraph at page 24, beginning at line 11:

If these weights were used then it would cost more to align differing tokens in the confined region than the offered region for example. This is because it has been confirmed and the input is thus less likely to be a correction of this region. Correction of the given region could ~~be~~ cost less because there has been no attempt to ground it yet.

Please amend the paragraph at page 27, beginning at line 8:

Then at Step 723, the next chunk is then set to span I2, and, the new remainder becomes simply R2 again by:

Please amend the paragraph at page 27, beginning at line 30:

Conceptually a CU represents a block of digits as input by the giver. That is to say in the simple case the region stretching from $\mathbf{f}_i$ to just before $\mathbf{f}_g$. However, inputs can themselves be interpreted as corrections or repetitions of information input in previous utterances. In this case the original correction unit may be retained, but altered, and maybe lengthened, by this new input.

Please amend the paragraph at page 28, beginning at line 33:

By way of example in the diagram (FIG. 7) block 0 has a correction unit, CU_0, spanning to the end of the token buffer. $\mathbf{CU}_1$ and $\mathbf{CU}_2$ are not present (i.e. zero), denoting the fact that a continuing input from the giver has only happened at the start of the first block.

Page 36, line 1, delete **Chapter V**

Please amend the paragraph at page 37, beginning at line 12:

"Thank you" could be added as an explicit completion signal immediately following the echo when the digits given so far constitute a complete number. This should work well, and would correspond to the most common pattern in human operator dialogues, when there had been no error and correction during the number transfer; but it could confuse the giver in the case where one of the blocks given was actually a replacement for the preceding block and the number was therefore not complete. The more subtle completion signal adopted in Figure ~~11~~2 Oust a change from continuing to ending intonation in the echo) seems less likely to cause such confusion. The "Thank you" message is given only after a completion signal or silence from the giver.

Please amend the paragraph at page 37, beginning at line 26:

The repair-from-end rule will also fail in cases with insertion and deletion errors ~~(artificially excluded from the trials to date, but occurring occasionally as wizard errors),~~ since it assumes that the correcting digits must replace the same number of digits in the previously recognised input. Here again, similarity-based matching might be required.

Please delete **Chapter VI** at page 37, line 31.

Please amend the paragraph at page 38, beginning at line 1:

The first--start(initial)--notes that is not essential that the original input that is now to be read back and confirmed by means of a spoken response should itself actually have been generated from a speech input. Thus the process shown in FIG. 9 could be applied to input from another source. One example of this, in the context of a telephone number dialogue might be where the system obtains a number (or part of a number such as the STD code) from a database, or perhaps assumes that the code will be the same as the user's own STD code, but needs to offer it for confirmation in case it has "guessed" wrongly.[["]] This can be done by using the start(initial) route and setting initial to the assumed `STD`. The giver can then confirm this, correct it, or continue to give the number in the same fashion described previously.

Please amend the heading at page 38, line 20:

~~Chapter VII~~ Dialogue, fourth versions

Please amend the paragraph at page 41, beginning at line 5:

An input is received from the giver. This input is then interpreted--with reference to the current token buffer--to decide which tokens in the buffer are to be updated by the new input. The interpretation has a cost of match associated with it. Once this interpretation is decided then the buffer is altered to give it new tokens in altered states. The cost of the buffer is increased by the match cost. Then the start and focal ~~indexes~~ indices for the giver are updated.

Please amend the paragraph at page 41, beginning at line 22:

The `repaired` category is tried when the grounded buffer does not match any of the patterns. In this case an attempt is made to repair the buffer by looking for other plausible token sequences given the dialogue history to date. If this is possible then the repaired buffer is matched against one or all of the grammars as above. If no match can be found to a repaired buffer then the process ends with a failure condition. If a match can be found for the repaired token buffer[[--]]then the token buffer is re-grounded to make sure that the repair was correct.

Please amend the paragraph at page 41, beginning at line 29:

After each turn which does not result in a conclusion the user's input is gathered and the cycle is repeated. This continues until a successful transfer occurs or the dialogue ends in failure.

Please amend the paragraph at page 42, beginning at line 28:

Determining the status of the token buffer is done by following the flow diagram shown in FIG. 12. The buffer is first tested 1201 to see what its grounding state is. If it contains any ungrounded tokens then it is considered **ungrounded** 1202. This means that the contents of the buffer has not yet been fully agreed between the giver and the receiver. This will usually be the case until the giver gives a completion signal--such as `yes` -- or remains silent at the end of a sequence of digit transfers. It can happen in other ways--the most special being the initial condition where the initial empty buffer is considered to be fully grounded and then matched against an empty pattern to kick the whole algorithm off. Ungrounded buffers are then further categorised into one of four ungrounded sub-states. These ungrounded sub-states are returned as the status of the token buffer.

Please amend the paragraph at page 44, beginning at line 10:

ungrounded(repeated)--The giver focal point ($\mathbf{f_g}$) is before the receiver focal point ($\mathbf{f_r}$) and the buffer contains no initial tokens ($\mathbf{1_A}$) or corrected tokens ($\mathbf{1_B}$) before the receiver focal point $\mathbf{f_r}$. This state typically occurs when the offered tokens to date have been correct but for some reason the giver has echoed some of them and stopped short of the point that the last offer reached. For example givers sometimes exhibit disfluent re-starts where they begin from the start of the utterance again even if the transfer is going well. In this case there is no need to correct anything, but the receiver must re-synchronise to the giver's new focal point and signal that it has done so. This is done by repeating the giver_s previous input (with additional prior context if necessary) up to the point where the input ended. See *repeatedChunk*( ) for details.

Please amend the paragraph at page 45, beginning at line 13:

A grammar definition is simply a representation of all possible token sequences which are valid for that grammar to be matched. At its simplest, each grammar definition could simply be a list of valid sequences. As is well known to those in the field, typically finite state grammars or context-free grammars are used as a compact way to enumerate valid paths. For example in the current embodiment of the invention regular expressions as used in the Perl programming language (these are finite state grammars)[[,]] are used to define valid token sequences for each grammar.

Please amend the paragraph at page 46, beginning at line 1:

no STD--There is a complete number body but no STD code. Prompt for the STD code. Invoke the function setInsert( ) to re-wind $\mathbf{f_o}$ and $\mathbf{f_r}$ to the start of the buffer (i.e. make them one), and temporarily adjust the cost of insertion of any input token after the start state to be zero. Also set the cost of inserting any input token after any token in the given state to be zero. See Table 7 and associated explanation for how this works. As a result of these temporary changes, the input [[it]]is inserted at the head of the buffer. The default costs are restored prior to the next giver input once the token buffer contains new tokens again.

Please amend the paragraph at page 46, beginning at line 18:

The function is trying to establish the best point before the first ungrounded token (i.e. in state $\mathbf{1_A}$ or $\mathbf{1_B}$) to start the output chunk. The end point of the next chunk to be output $\mathbf{f_r}$ is always set to the last giver focal point $\mathbf{f_g}$. i.e. The receiver re-synchronises its focal point with the giver's focal point. There are three sub-conditions which may be observed in the corrected chunk state. These define the choice of start point according to which of the following ordered list of conditions is found to be true the first:

(a) **First token in state 1B or 1A is found on or after $\mathbf{f_o}$.** The first ungrounded token is ~~before~~at or after the start of the chunk that the giver has just heard. In this case keep the start point of the next offered chunk the same for the next offer[[.]] (i.e. f_o remains the same).

Please amend the paragraph at page 47, beginning at line 32:

The function *prependedChunk(tokenbuf)* is used at Step 1116 when the token buffer is found to have the status **ungrounded(prepended)**. The function decides the start ($\mathbf{f_o}$) and end ($\mathbf{f_r}$) points in the buffer for the buffer token sequence, or `chunk`, to be offered back to the giver with [[a]] `ok` pre-pended to it, using the *output*() function.

Please amend the paragraph at page 48, beginning at line 24:

b) **fo>f1**. That is to say the previous input wound-back before the previous chunk. In this case adopt the token immediately after the phrasal boundary before the giver start point ($\mathbf{f_i}$). Otherwise, if there ~~are~~is no phrasal boundary before the first correction it re-starts from the start of the buffer.

Please amend the paragraph at page 49, beginning at line 21:

After setting the pointer in Step 1114, 1116, 1118 or ~~1112~~1120, the system makes (except in the continued case) an appropriate announcement.

It then uses the *output*( ) function 1128 to play back to the giver the tokens contained between the modified receiver pointers. It also sets these token states to be $\mathbf{F}_A$ if they are not already $\mathbf{F}_B$. That is to say [[is]] marks all of the offered items to the highest grounding state that is appropriate.

Please amend the line 32 at page 49:

/*Choose ending or continuing intonation ~~foe~~for end of chunk */

Please amend the paragraph at page 50, beginning at line 24:

Following output, the function *modifyCP(tokenbuf)* 1129 (described below under "Final Repair") is invoked and the flow then proceeds to 1108.

Please amend the paragraph at page 51, line 13:

Then an appropriate announcement is made at Step 1132. Logically the process should now proceed to re-evaluate the status of the token buffer at 1104. However for all non-empty ok match grammars the outcome of this test will always be ungrounded(continued). Thus for simplicity the process proceeds directly to step 1120. This starts at the start of the buffer again, and the process of progressive read-back and confirmation of the repaired token buffer is initiated afresh. In the case of the other various match conditions, an appropriate announcement is made (1134,~~1136,~~ 1138, 1140) and the process either terminates (1142, 1144)or proceeds to get(input) 1108.

Please amend the paragraph at page 51, beginning at line 21:

Once the system receiver response has been decided, the giver's response is recognised at step 1108 *(get(input))*. This recognised input is stored in a buffer similar to the token buffer where each input token has a value and a state. Initially these inputs will all have state `U` and the process of getting the values for this input buffer has already been described previously.

Please amend line 10 at page 52 to read as follows:

return.token=concatenate(**return.token,"")**

Please amend the paragraph at page 53, beginning at line 5:

As can be seen in FIG. 11, the following cases are detected:

(a) Unclear digits. (e.g. "3 ? 4" with the middle digit being difficult to hear). A "sorry?" prompt is played at ~~1148 [Not the right number??]~~1146 to prompt for a repetition of this block. `*pardon*?` would be similarly effective.

Please amend the paragraph at page 53, beginning at line 17:

(e) *Contradiction and digit correction--possibly self-repaired digits* (e.g. "no 3 4 5"). If the input itself contains a self-repair (e.g. "no 0 4 no 0 1 4 1"), it will first be repaired at 1110 using the localRepair algorithm described earlier. The fact that it is a clear contradiction will be noted, then the input will be optimally aligned against the current state of the token buffer and the buffer state updated using the ~~intepretInput~~interpretInput process 1112, to be described below.

Please amend the paragraph at page 53, beginning at line 26:

(g) *Completion signal (silence or "yes")*. All offered, but unconfirmed tokens ($\mathbf{F_A}$ or $\mathbf{1_B}$) in the buffer before the receiver focal point are set to the confirmed state ($\mathbf{F_B}$), and the given start point ($\mathbf{f_i}$) and focal point ($\mathbf{f_g}$) are set to be the same as the receiver focal point ($\mathbf{f_r}$); the process proceeds to 1154.

Please amend the paragraph at page 54, beginning at line 10:

The function *confirmToFocus(tokenbuf)* at Step 1154 is called when the giver has said an isolated "yes," or remained silent following a set of tokens being offered. It sets all token states from the start of the offered region ($\mathbf{f_o}$) to the receiver focus to the `confirmed` state $\mathbf{F_B}$. Also, the giver start point $\mathbf{f_i}$ and focal point $\mathbf{f_g}$ are both set to the receiver focal point $\mathbf{f_r}$. This is because it is assumed that the giver has adopted the receiver focal point when remaining silent or explicitly saying `yes` after the focal point.

Please amend the paragraph at page 54, beginning at line 23:

It should be noted that this embodiment of the invention may be used for tasks other than digit entry. If the symbol `D` in FIG. 11 is taken to mean `the grammar of legal tokens in this task`, then the conditions above would apply to any token transfer task. Appropriate completion criteria along with grammars would also have to be designed for a new task. It is however likely that ~~the~~ similar patterns for completion will be observed in many sequential token transfer tasks.

Please amend the paragraph at page 56, beginning at line 18:

Because insertions and deletions are possible, it is necessary to note the changes to the receiver focal point and receiver start point which will need to be applied should this input be accepted. The receiver focal point will change if tokens are inserted or deleted prior to the current receiver focal point. The receiver start point will only change if tokens are inserted or deleted prior to the current receiver start point. Any other stored indices which the algorithm relies on (for example see Correction Points as described later) will similarly need to be adjusted once a particular interpretation is adopted. Pointers to deleted elements in the buffer will be assumed to point to the token following ~~to~~ the deletion as part of this adjustment.

Please amend line 22, page 57 to read as follows:

tokenbuf.cost +=$\mathbf{d}_{k[[,]]}$

Please amend the paragraph at page 57, beginning at line 27:

A dynamic programming alignment algorithm essentially takes two sequences A, B of tokens and evaluates possible alignments of them to find the one representing the best match between them. More accurately, the algorithm ~~established~~establishes the optimal way to convert one sequence of tokens A into the other sequence of tokens B given a set of costs for deleting, inserting, and substituting tokens.

Please amend the paragraphs at page 58, beginning at line 16:

Once the sequences have been aligned in this way a measure of similarity between them (representing, in the current context, a cost of substituting one for the other) can be calculated. For example if the cost of aligning equivalent tokens is zero, a substitution of one for another is 1 and the cost of a deletion or insertion is 2 then in the first example the cost is simply that of inserting the `5` into A--a cost of 2 units. In the ~~second~~third case the cost is that of two insertions, one deletion and one substitution, i.e. 7. Note that these values are for illustration only; also this is a symmetrical example for the purposes of illustration and that for present purposes the cost rules are more complex, as will be described shortly.

All possible alignments may be visualised by a graphical representation as in FIG. 16. A is shown along the vertical axis and B along the horizontal axis. The small squares represent nodes: the arrows represent all the possible paths through the nodes, from the top left ~~top~~to the bottom right. Traversing a horizontal arrow corresponds to an insertion from B into A--or copying the B-value for that column into B', and a null into A'. Traversing a vertical arrow corresponds to deleting a token from A--or copying a null into B' and copying the A-value for that row into A'; and traversing a diagonal arrow corresponds to a substitution--or copying both A and B into A' and B' respectively. The cost for any path entering a node is the cost associated with the preceding node plus the cost associated with the path itself. The cost associated with any node is the minimum of the costs associated with the three paths leading to that node. Costs for the example are shown on the diagram. The DP algorithm works through the nodes to find the costs whilst at the same time recording the paths that correspond to the cheapest route into a node. It then backtracks through the nodes from the bottom right and determines the path that corresponds to the minimum cost. The path shown in heavy lines is thus the optimal path. It corresponds to the first example given above.

Please amend the paragraph at page 59, beginning at line 8:

In the DP alignment a table of specific costs is used in order to evaluate the relative costs of various substitutions, deletions or insertions. Table 7 shows some example costs which could be used for a digit-only entry dialogue such as a telephone number entry. In the table some conventions are used. Unlike the algorithm described in our aforementioned [[']]international patent application, insertions and deletions are not symmetrical in this task. The sense of the terms *insertion, deletion and substitution* (ins, sub, del) are used to refer to what changes would have to be made to the token buffer in order to transform it into the input buffer. The term `Eq` is also used to represent a special case of substitution--the substitution of a token with another token of the same value. FIG. 14 shows the sense of the operations graphically with respect to the DP alignment cost matrix.

Please amend the paragraph at page 59, beginning at line 24:

Unlike an ordinary DP match however, contextual information is also used to select between different specific costs for deletion and insertion. FIG. 14 shows which tokens are taken as the context during the DP cost calculation. The effect of these contexts may be interpreted as follows. In the case of the **deletion** of a token in the buffer, it is the state and value of the input token that aligns with the token that is just before the deletion, and the state and value of the buffer token being deleted which are used to decide the context. In ~~this~~the case of **insertion** of an input token into the token buffer, it is the state and value of the input token being inserted and the state and value of the buffer token after which it will be inserted which are used to decide the context. For

substitution,~~ the context of~~ the two tokens under consideration for the substitution are used as the context.

Please amend the paragraph at page 61, beginning at line 8:

Secondly, it can be seen that there is no cost at all for substituting ungiven values. This simply means that freshly given tokens get appended to the end of the buffer at no cost if there are no previously given ~~token~~tokens to be substituted. The cost of substituting a given or offered token with a different one is unity (1.0) rising to a higher value if the token being substituted is fully confirmed. This means that confirmed items can still be corrected, as it has been observed that givers may mistakenly confirm erroneous inputs, but interpretations for correcting tokens in lower grounding states are preferred.

Please amend the paragraph at page 62, beginning at line 11:

Recall that DP alignment fully aligns both the input buffer and the ~~potion~~portion of the token buffer selected by variable **k**. This is not always desirable in this case. For this reason the cost structure is set up such that when the final meaningful token in the input has been aligned in the DP search, the final dummy input token in the `S` state will insert optimally immediately following it. This token is then permitted to delete remaining tokens in the token buffer up to the end point--freely if the tokens are ungrounded or at a cost if not. This mechanism allows the end of the input to align with tokens other than the end of the tokens in the token buffer if this gives an optimal fit. This is appropriate especially if the remaining tokens in the buffer are still in state "$\mathbf{1}_A$"--

comprising the `remainder` seen in previous embodiments of the invention. When interpreting this alignment the algorithm defines the end align point of the input to be where the `S` state was inserted in the optimal trace *and retains all of the deleted tokens after this point in the token buffer.*

Please amend the paragraph at page 62, beginning at line 24:

It was noted previously that it is also possible to accommodate phrasal boundaries in the buffer, and also in the input. Putting phrasal boundaries in the buffer is useful for two primary reasons. Firstly, it permits the system ~~use~~to adopt the chunking strategy used by the giver (mid utterance phrasal boundaries if these can be detected by the input device) when reading back the number to the receiver. This can be seen in the operation of continuedChunk( ) which uses the phrasal boundaries to determine whether the whole or part of the ungrounded material left in the buffer should be read out for confirmation next.

Please amend the paragraph at page 62, beginning at line 31:

Secondly, the boundaries are used to bias the alignment of inputs to the buffer. This supports the observation that givers and receivers co-operate together to use common phrasal boundaries to remain in synchronisation. However ~~is~~it also supports divergence from these patterns if the receiver or giver should choose to do so. Table 8 shows a set of costs which will support this functionality. The principles underlying this table are that a phrasal boundary should never delete a token, and that in general phrasal boundaries should align between the token buffer and the input, but where they don't, both phrasal boundaries are retained. This has the potential to proliferate phrasal boundaries in certain circumstances, namely where the giver and follower do not accept the same phrasal boundaries. In these circumstances the algorithm will tend to offer smaller chunks of output tokens.

Please amend the paragraph at page 64, beginning at line 8:

As an alternative to the use of special tokens to signal the phrasal boundaries, one could instead--with, of course, corresponding modification to the DP algorithm[[,]] -- store these boundaries as additional state information with an additional storage location for each token being provided for this purpose, as illustrated in FIG. 15.

Please amend line 25, at page 64 to read as follows:

foreach k (0..length(input)-1 {

Please amend the paragraph at page 65, beginning at line 11:

Final repair (1205, FIG. ~~10~~12) in the fourth embodiment of the invention is similar to that shown in previous embodiments but is simplified. It also uses the same cost based dynamic programming alignment mechanism as has already been described. In this simplified usage, Correction Units (CU's) are non-overlapping regions of the token buffer between Correction Points (CP's). Recall that previously CU's could overlap. CP's are simply pointers into the token buffer to a point where a pure continuation was decided upon following an interpretation of the input. Recall that, as in previous embodiments, at these points the giver cannot tell whether their input has been interpreted as a continuation or a correction due to the simple echo strategy employed.

Please amend the paragraph at page 66, beginning at line 8:

For the special case when tokens are ~~insertion~~inserted at the start of the buffer--this should be treated as a pure continuation for the purposes of the CP algorithm. This will occur for example following the temporary adjustment of costs as a result of the setInsert( ) function. A new CP at index 1 should be set. The CP that was at index one is retained at its new right-shifted position. The CP is retained because the current cost structure associated with *setInsert*( ) always inserts at zero cost--thus if the insertion partly contains a continuation into the shifted material this will not be captured at that point. However if we treat the sequence following the insertion as if it were a possible ambiguous correction of the preceding tokens then this wrong assumption can be repaired in the final repair stage.

Please amend the paragraph at page 66, beginning at line 22:

Following a failure to match any grammar in a grounded token buffer. Final repair is attempted at ~~1125~~1205 by the repair( ) function. This takes a token buffer with its correction points, and attempts to find a new interpretation which matches one of the grammars (`ok` being the preferred match) with the lowest possible additional cost.

Please amend the paragraph at page 66, beginning at line 26:

The final repair process is a stack based operation. Initially the stack is populated, in any order, with a number of copies of the token buffer. In these initial copies different permutations of the CP's are retained or discarded to represent all possible permutations of the existing CP's--excluding the case where there are no CP's (~~This~~this has already failed the match test and is therefore not worth exploring). In the original buffer CP's represented *possible* correction points. In these new hypotheses, the presence of a CP indicates that for this hypothesis there is *definitely* a correction at that point. The initial CP is treated as a special correction point and is always retained. Thus, if the token buffer contains NCP correction points then there will be $(2^{NCP-1}-1)$ copies of the buffer on the stack. Previous embodiments of the invention only allowed the equivalent of one correction per buffer. This embodiment allows multiple corrections if they have a low cumulative cost. If the previous behaviour is to be emulated, then this step could simply create NCP-2 hypotheses instead where each hypothesis contains only one correction point plus the initial correction point.

Please amend the paragraph at page 67, beginning at line 12:

The algorithm pops hypotheses off an input stack until the stack is empty. The order in which repair occurs at the CP's in this hypothesis is important. There are N factorial (N!) evaluation orders for a hypothesis with N active CP points. The algorithm evaluates all of these possibilities by stepping through the active CP's in the current hypothesis and selecting each ~~CP's~~ CP as the start point. An optimal repair is made at this point by aligning the CU after the CP with the CU before the CP. The same *interpretInput*( ) algorithm is used that was used in the live dialogue based repair. In this way the *finalRepair* algorithm could be seen to be evaluating the options which were second best when interpretations of k=0 were selected in the live dialogue.

Please amend the paragraph at page 67, beginning at line 21:

By repairing at the site of this CP, it is eliminated as a result and the resulting new hypothesis now contains N-1 CP's in it. If there are remaining active CP sites then this new hypothesis is placed back on the stack for evaluation at a later date. By following this procedure for all possible CP start points then a hypothesis with N active CP's will generate N new hypotheses on the stack with (N-1) active CP's in place of the original hypothesis. By evaluating the stack until it is empty this will create the necessary N factorial (N!) evaluations to explore all possible start points.

Please amend line 10 at page 69 to read as follows:

/*Find lowest scoring ~~hypotheses~~hypothesis which matches ok grammar*/

Please amend the paragraph at page 69 beginning at line 32:

The following pseudo code detects and removes and returns [[an]]a UK STD code at the start of a block. NB STD code patterns can change fairly frequently in the UK due to strong regulatory involvement.

Please amend the paragraph at page 71, beginning at line 12:

The following function plays a chunk of digits out imposing an appropriate intonation on the chunk to make it sound natural. If endInton="ending" then the intonation of the final digit of the chunk will signal that there are no more chunks to follow (e.g. signal that the dialogue believes that the last chunk in the telephone number has been received). If ~~endInto~~endInton="continuing" then the final digit will have intonation which indicates that further digits are to follow in a subsequent chunk.

Please amend line 7 at page 74 to read:

```
e=SUM(B(n-k)...B(n-1))
```

Please amend line 11 at page 75 to read:

```
IF e>L(n) NEXT k
```